CMU-CS-80-134

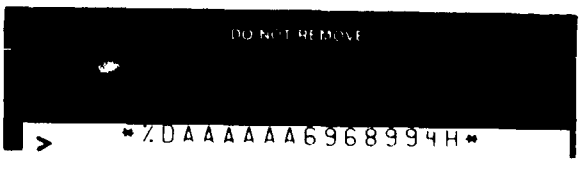
The Design of a Network-Based Central File System

Mike Accetta
George Robertson
M. Satyanarayanan
Mary Thompson

August 1980

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University



CMU-CS-80-134

The Design of a Network-Based Central File System

Mike Accetta
George Robertson
M. Satyanarayanan
Mary Thompson

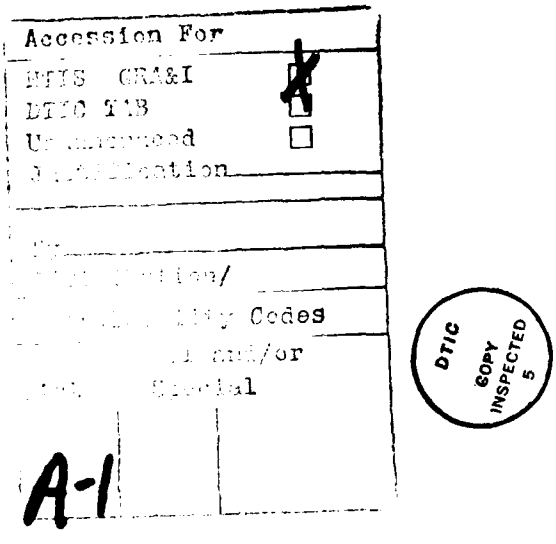
August 1980

Abstract

The file system described here will be the primary means by which a variety of computers connected together by a local network will share files with each other. The spectrum of computers using this network will range from mainframes supporting time-sharing to personal computers. The services provided by the central file system include the storage, migration and archiving of files, the naming and sharing of files and directories, and the authentication of users. This document presents an overall view of the services provided by the facility and describes the primitives used in accessing it. The system described here is being implemented in the Department of Computer Science at Carnegie-Mellon University. However, the design is intended to be applicable to any installation whose computers are linked by a local network, and which possesses a large-capacity archival storage system.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents



1. Overview

1.1 Introduction

The *Central File System* (CFS) has been designed as a back-end file system for the computing environment in the Department of Computer Science at Carnegie-Mellon University. This environment currently consists of a variety of machines such as PDP-10s [2], VAX-11/780s [3], PDP-11s [4] and Altos [13] connected together by a local network [8]. The Spice [12] machines currently under development will be added to this environment when they become available. The file systems supported by the operating systems on these computers span a wide spectrum.

The CFS will provide a central facility on which local file systems may store files and from which they may retrieve them. Files on the CFS are accessible to all machines on the local network. Besides permitting sharing of files, the CFS will also provide a means of indefinitely archiving the current and past versions of all such files.

The major use of the CFS will be made by the various local file systems acting on behalf of their respective users and not generally by any particular user directly. The CFS does not distinguish between human users and programs which use the CFS just as human users would. Consequently the word "user" will be used in the rest of this document to designate any entity that uses the CFS.

The design of the CFS, while unique as a whole, draws upon ideas from a variety of existing file systems. The directory structure is based on that of the UNIX file system [11]. The notions of stable storage and transactions are based on work done at Xerox [7, 6] and IBM [5]. The Multics [9] and Hydra [1] file systems have also had an influence on the design of the CFS. Finally, the promise of extremely high storage density write-once media such as video disks [15] has influenced the archival and migration features of the CFS, and encouraged the notion of invariant files.

The prototype system will be implemented on a VAX under the UNIX operating system but completely distinct from the existing UNIX file system on the machine. The first two file systems to be interfaced to the CFS will be the VAX UNIX file system and the Spice file system [14] with the plan of eventually connecting all of the machines in the environment. The two initial local file systems will make use of the CFS in slightly different ways. UNIX will treat the CFS as a central storage area for common system files and as the primary backup and archiving area for general files. The Spice file system, on the other hand, is being designed with the intent of providing the Spice user with a local file system which resembles the CFS as closely as possible.

1.2 Functions of the CFS

The CFS provides users with a wide variety of services. The degree to which these services are used is left completely up to local file system implementations. CFS services will be provided over the network by three server processes: a *File Server* which provides access to the physical data in files, a *Name Server* which supports a directory structure, and an *Authentication Server* which handles the initial connection protocol between users and the CFS, and establishes the authenticity of the former. Also present, but invisible to users, will be an *Archive Server* which interacts with the File Server to handle automatic archiving and migration functions. The Interprocess Communication facility (IPC) [10] will be the sole means of communication between users and the CFS, and between the components of the CFS.

At the lowest level, the CFS will provide the common storage mechanism for files from all users. It will support files with differing data characteristics (such as text, and 8-, and 36-bit binary), to simplify the sharing of compatible data between a variety of local file systems, and will provide a central location for the storage of mailboxes, bulletin boards, documents, data bases, and other shared files. It will also provide a common archiving mechanism for retaining all files in perpetuity and for recovering any file through normal file access operations.

A noteworthy feature of the CFS is that it permits simulation of file accesses by processes. As described in section 2.2.1, users may request file operations such as read and write on the simulating process. The fact that the "file" is really a process can be made totally transparent to users by ensuring that the process responds appropriately to all file access requests.

The Name Server provides a service which will primarily be used for naming files. However, it is possible to use it as a mechanism for naming entities other than files. For example, it can be used to provide system-wide names for IPC ports. This use of the CFS is described in more detail in section 2.3.2.

At a higher level, the CFS may be viewed as an extension to each local file system. The standard operations of creating and deleting files, reading and writing file data, updating file statistics and changing access rights are all basic functions of the CFS. They may be combined by the local file systems in whatever manner is convenient, extending from completely caching local copies of CFS files through modifying files in place on the CFS. The functions of a local file system can be integrated with those of the CFS in a manner which presents the user with a single, homogeneous interface.

The CFS supports concurrent sharing of files via *transactions*. All of the read and write requests within a transaction are collected together such that any changes are made indivisibly as a single unit when the transaction is completed. The transaction mechanism is primarily intended for synchronizing concurrent updates in shared file applications.

1.3 Communication with the CFS

All communication with the CFS is accomplished through IPC messages between authenticated ports on the CFS and host machines. As explained in the IPC document [10], a port is essentially a queue of messages with many senders but only one receiver. Most of the ports used in CFS will have only one sender and one receiver. An authenticated CFS port is one for which the identity of the senders and the receiver is established to the satisfaction of the CFS. Network security guarantees that a port remains authenticated between the initial verification and the deallocation of the port.

A message to the CFS consists of one or more typed parameters which specify the service being requested. Messages are passed back and forth across the network via network IPC servers on each machine, and are passed on to user processes by local IPC mechanisms. This is done in a manner which completely conceals the fact that a destination or source port is actually on a remote machine. In addition, any network security and authentication necessary to prevent the forging of access to remote ports is ultimately the sole responsibility of the network servers and not of any particular user process.

By using such a transparent network IPC mechanism, the CFS presents the user with a uniform interface, independent of specific network topology and distribution of CFS functions. User processes view the CFS simply as a collection of other local processes and communicate with them via the local IPC mechanism. The actual CFS services may be distributed with reasonable ease across multiple processes, machines, and networks as the need arises, and this can be done completely without the knowledge of users.

2. General Design

2.1 Authentication Server

2.1.1 Establishing a connection

In order to use the services of the CFS, a user must first establish a secure connection with it. This connection is established by sending a login message, consisting of a CFS user name and password, to a public port of the Authentication Server. Depending on the local file system, this login connection may be established explicitly at the request of the user or implicitly when a user logs in to the local machine. As a result of a successful login, three authenticated ports are established and returned: one to the File Server, another to the Name Server, and the third to the Authentication Server. These ports are used for all further communications with the CFS.

The connection to the CFS is terminated by sending a logout message to the Authentication Server port. This deallocates the File, Name, and Authentication Server ports and closes the connection in an orderly manner. This may involve closing open files and aborting transactions. The CFS connection may also be broken implicitly by events such as a time-out due to local machine crash. In these cases too, the connection is terminated appropriately.

2.1.2 Access Groups

Protection in the CFS is enforced via the notion of *access groups*. An access group defines a subset of users to whom privileges may be collectively awarded or from whom these privileges may be collectively revoked. Access groups are partitioned into two classes: *primary access groups* and *secondary access groups*. Each user of the CFS is the sole member of exactly one primary access group. Since there is a one-to-one relationship between primary access groups and users, the terms "user" and "user's primary access group" will be used interchangeably in this document. Secondary access groups are created by users and differ from primary access groups in two ways:

1. Secondary access groups may not login to the CFS, and

2. A secondary access group may contain more than one user.

A user may belong to any number of secondary access groups, and at any instant of time, the protection environment of a user is defined by the union of the rights of all the access groups of which he is an *enabled* member at that instant. The notion of *disabled* member is used to permit a user to deliberately deprive himself of certain rights: for example, while debugging a file manipulation program, a user may not wish to have *Write* privileges on a sensitive file.

Associated with every access group is a CFS data structure called the *group attribute file* or GAF (see appendix IV). The GAF contains all relevant information about that access group, and is uniquely identified by an integer called the *access group ID.*

Every access group also has a name that identifies it uniquely. In the case of a primary access group, this is the name by which the corresponding user logs in to the Authentication Server, and in the case of a secondary access group, this is the name by which that access group is referenced. Access groups may be renamed; however, their IDs cannot be altered. Every user is automatically a member of the group called ALL, which consists of all CFS users.

When a user logs in to the CFS, each of the IPC ports returned to him has associated with it a list containing all the access groups of which the user is an active member. Enabling or disabling of access groups by the user causes these lists to be appropriately modified. As described in sections 2.2.3 and 2.3.4, the Name Server and File Server use these lists in order to control access to files and directories.

While primary access groups are restricted to having exactly one member, there is no restriction on either the number or type of members of secondary access groups. One implication of this generality is that it is possible to construct a hierarchy of secondary access groups. This is done by defining a secondary access group to consist of a set of other secondary access groups (or a mixture of primary and secondary access groups). Thus, if A and B are two access groups, one can define C to consist of these two access groups. Every member of either A or B is automatically a member of C. Adding or deleting a member from A or B thus implicitly adds him to C or deletes him from C. The relationship *"is a member of"* is thus a transitive one. Consequently, the reflexive transitive closure[1] of this relationship applied to any access group yields a list containing every member, direct or indirect, of that group. When a set of access groups is enabled by a user, the union of the corresponding transitive closures yields the list of groups to be used in access checks. Thus, the set of rights that a user has on any entity is the union of the rights that his currently enabled access groups have on that entity.

This construction of the transitive closure (referred to as "flattening") of a user's access list is done at login time and any time a user enables or disables an access group. Thereafter, the flattened access list is associated with the ports to the File Server, Name Server and Authentication Server.

Primary access groups can only be created and deleted by the System Administrator whereas secondary access groups can be created and deleted by users. An access group's attributes are protected against unauthorized modification by an access control list mechanism. Each access group has an access control list associated with it. This list consists of a set of access group IDs and the rights that each of them has on that access group. The set of access rights are:

---

[1] For brevity, the phrase "transitive closure" will mean "reflexive transitive closure" in the rest of this document.

AddMembers members may be added to the group

RemoveMembers members may be removed from the group

DeleteGroup the group may be deleted

GetGroupAccess the access control list for the group may be read

SetGroupAccess the access control list for the group may be modified

2.2 File Server

2.2.1 File organization

A file is stored on the CFS as an ordered collection of *data units* divided into *pages* containing a fixed number of these data units. The size of each data unit as well as the number of data units per page depend on the data characteristics of the file (see section 2.2.2 and appendix III). The CFS supports file operations at the grain of data units. However, operations which deal with whole pages are likely to be more efficient. A file may contain *holes* corresponding to data pages which have never been written. Attempts to read data from a hole are detected and reported to the requesting user. These pages are not actually allocated storage on the CFS.

To permit efficient storage and retrieval of data from the archive, an additional structure is imposed on files. A contiguous set of pages which may be stored in or retrieved from the archive independent of the rest of the file is called an *archival unit* (abbreviated *archit*). There are no a priori restrictions on the size of archits. As a consequence of the fact that files may contain holes, it is possible for the archits in a file to be of the different sizes. It is expected that for most small files without holes, the entire file will be a single archit. Larger files will typically consist of a number of archits, their maximum size being specified by the creator or suitably defaulted by the CFS. It should be noted that archits are present purely for efficiency. A user may choose to completely ignore the fact that a file really consists of a set of archits. The CFS maintains information regarding the dates and times of last modification and access of each archit in a file. Depending on the file usage pattern, it is quite possible for some parts of a file to be present on secondary storage while others are only available on archive --- this fact is completely transparent to the user.

A file may be stored in either *standard storage* or *stable storage* [6]. The data pages of a file stored in standard storage are recorded in only one place on the CFS while the data pages of a file stored in stable storage are recorded in two different places on the CFS. Files stored in stable storage are less likely to be damaged during a system crash than are files stored in standard storage, however, they are also more expensive to store and change. Consequently, most files will be stored in standard storage. Only extremely important files will be stored in stable storage. The storage locations of stable pages are chosen so as to reduce

the chance that both copies can be damaged at the same time (e.g. on different platters of a disk pack in case of a head crash). The two physical copies of any stable logical page are always written by the File Server in the same order so that they can be brought into agreement after a crash when they are not identical.

All I/O operations on files are expressed in terms of the position, in units of file storage, at which to begin the I/O and the number of units to transfer. I/O which extends beyond the current size of the file adds new data to the end of the file on writes (depending on the access permission) but is disallowed on reads.

Every file is uniquely identified by its *file ID* (FID). The FID is a unique integer which is generated by the CFS when the file is created and is never changed thereafter. FID's are the only means by which files are identified to the File Server. A given FID always refers to the same file (although its data may change), and is never reused.

The File Server accepts an IPC port in lieu of an FID. All file access requests for that "file" are routed to the port by the File Server. It is the responsibility of the process receiving on that port to interpret these requests and to act accordingly. Replies from the process are sent directly to the requestor. This is the mechanism by which the File Server supports the notion that files may be active entities (i.e., processes) rather than passive information structures.

2.2.2 File Properties

Files stored in the CFS are described by four properties, each of which is orthogonal to the others. These are:

Mutability

Possible values: {*Variant, Invariant*}

The *mutability* of a file determines whether the data in that file can be altered after file creation. An *invariant file* can never have its data altered whereas *variant files* can be so altered. In practice it is expected that the majority of files will be invariant. A series of (logical) modifications to an invariant file will be reflected as a series of invariant file creations. Thus, the familiar actions of reading a file, editing it and overwriting the original copy will be accomplished by reading an invariant file, editing it and creating a new invariant file corresponding to the edited version of the original.

The version mechanism provided by the name server (see section 2.3.3) will bear the burden of keeping track of the invariant files corresponding to successive versions of a filename. The use of the name server is, of course, optional --- local file systems and higher-level mechanisms on host computer systems may choose to keep track of versions themselves.

Variant files are intended primarily for use in database applications, where it is inappropriate to recopy the entire file when modifying it. Concurrent updates of such files are possible and mechanisms to guarantee the indivisibility of updates are provided by the CFS.

Both the CFS and the local file systems can make many simplifying assumptions about invariant files. Consequently, some operations on invariant files will be more efficient than the corresponding operations on variant files.

Storage Characteristics

Possible values: {*Standard, Stable*}

This property describes how well a file is protected from hazards such as disk crashes. Since files are archived soon after their creation on the CFS, standard storage is adequate for most purposes. In case of a disk crash, lost files can be recovered from the archive. But because there is a finite delay between the creation of copies on disk and archive, a disk crash which occurs between the creation of the two copies could cause the unrecoverable loss of a file. Particularly valuable files should therefore be stored in stable storage.

Data Characteristics

Possible Values: {*Universal text, 8-bit binary, 16-bit binary, 32-bit binary, 36-bit binary*}

The CFS will be connected to a variety of machines with different word sizes, and different software conventions regarding text files. By suitably setting the data characteristics of a file, a user can express operations such as read and write in terms of data units that are meaningful to that file rather than expressing them always in units such as bytes. The mapping of these data units into the internal storage format of the CFS is totally transparent to the user. Appendix III specifies the number of data units per page for different data types.

Advisory File Type

Possible Values: { any of a set of user-defined types }

This property is primarily for the use of higher-level mechanisms in host computers and corresponds to the notion of types in programming languages.

It should be noted that the *type* of a file is not of interest to the CFS --- it is present primarily for use by local file systems and higher level mechanisms on local computers. The allocation of types is not the responsibility of the CFS. The installation using the CFS has to have some external mechanism that manages the allocation and deallocation of t[illegible] and prevents conflicts. One suggested way of handling this is to have a special directory on the CFS to which all users have *LookUp* and *CreateName* access. To create a new type, a user could create a file and enter a description of the type in that file. He could then try to enter the FID of this file as version one of the type name in the special directory. If the *EnterName* succeeds, the user has effectively allocated himself a new type; the FID of the type description file becomes a unique identifier for the type. If the enter fails, the user knows that the type name has already been allocated. He may examine the corresponding type description file to see if that type meets his requirements. If it does he may use that type, otherwise he can choose a new name and try again.

2.2.3 Access control

Corresponding to each file is an associated *access control list*, which is a mapping from access groups to access rights on the file. When a user requests an operation on a file, the File Server examines the entries corresponding to each of the user's enabled access groups in the access control list of that file. If any of these access groups possesses sufficient rights to perform the requested operation on the file, the access check succeeds and further processing of the request occurs. Otherwise access is denied.

The access rights for a particular access group are defined by a collection of bits describing the operations which may legally be performed on the file by members of that access group. The set of access bits for a file is divided into *system access bits* and *auxiliary access bits*. Logically, one can view a file as having two access control lists: system and auxiliary. System access bits are interpreted by the File Server. Auxiliary access bits are stored on the CFS but normally have no meaning to the File Server; they are interpreted only by the programs or subsystems using them. The Name Server and the Authentication Server use auxiliary access bits on directories and GAFs to implement their own access checks.

The system access bits are:

- **Read** existing data may be read
- **Write** existing data may be written
- **Extend** new data may be added to the end of the file
- **Truncate** existing data may be removed from the end of the file
- **Purge** the file storage may be physically deallocated
- **ReadHeader** most fields of the file header may be examined
- **WriteHeader** selected fields of the file header may be modified
- **GetSysAccess** the system access control list may be examined
- **GetAuxAccess** the auxiliary access control list may be examined
- **SetSysAccess** the system access control list may be modified
- **SetAuxAccess** the auxiliary access control list may be modified

2.2.4 Synchronization

The CFS provides one implicit and two explicit mechanisms for synchronizing file operations.

The first explicit mechanism, supported for both variant and invariant files, is an *Advisory Semaphore* associated with each file. As the name suggests, this semaphore is purely advisory in function. The CFS

manipulates it in response to user requests but does not heed the semaphore in any other way. The primary purpose of this semaphore is to permit higher level mechanisms in host systems to synchronize their file operations. When the advisory semaphore on a file is set, the ID of the user requesting this operation is returned. Only this user may reset the advisory semaphore. When setting the semaphore, a user must specify the maximum length of time for which it should remain set. If the semaphore is not reset within this interval by the user, the CFS automatically resets it. The CFS does not provide for queueing of requests to set a semaphore. If a requested operation on a semaphore cannot be immediately carried out, it is up to the requestor to negotiate the release of the semaphore with the current holder.

For variant files, a second explicit synchronization mechanism is provided. A series of reads and writes to a subset of the pages of the file may be grouped together in a *Transaction.* The CFS guarantees that a transaction is executed as one indivisible unit. Modifications to a file, requested within a transaction, are not carried out until the transaction is ended. Thus, a transaction may be aborted any time before it is ended, without any changes to the file. Transactions may be read or modify (i.e., *write, extend or truncate*) transactions and take a list of pages as an argument. At any instant there may be at most one modify transaction or multiple read transactions per page --- the CFS ensures this by comparing the page list supplied at the start of a transaction with the page lists of transactions in progress. There are no time-outs on transactions; a transaction is terminated only when it is explicitly ended by the user or when the connection between the user and the File Server is broken.

A file has to be opened for transactions before read or modify transactions may be initiated on it. Since variant files can be modified outside of transactions (direct writes on the entire file are possible), the *Open* primitive has to perform implicit synchronization if consistency is to be ensured. This is best described by the following table:

The entry in row *i*, column *j* is an X iff an operation of type *i* will fail when the file has been opened by another process for operation *j*.

R = Open for Read
M = Open for Modify; i.e., for write, extend or truncate
R' = Open for read transactions
M' = Open for modify transactions;
i.e., for write, extend or truncate transactions

	R	M	R'	M'
R		X		X
M	X	X	X	X
R'		X		
M'	X	X		

A failure on open can either result in a queuing of the open request by the CFS or a failure code return, depending on parameters specified in the open request by the user process.

2.2.5 File headers

All information stored by the CFS for a file is recorded in its *file header* which is kept in stable storage. This includes everything from the file owner and access statistics to pointers to the actual data pages of the file. A file is accessed through its file header, which in turn is located by using the FID as a virtual index into the table of file headers for the entire system. All operations on a file are performed based on the information stored in the file header. The primitives provided by the File Server are those which create, modify, and delete file headers and the files they represent.

A file header consists of a series of fixed and variable length fields. Some of these fields are directly modifiable by user processes while others are only implicitly modifiable by the File Server on behalf of its users; appendix I specifies which fields may be modified directly by users. Most fields may be examined by user processes although some are internal to the File Server and completely hidden from external view. A file header contains the following fields:

File ID (for redundancy check and file system crash recovery).

File properties the four properties that characterize this file (see section 2.2.2).

Size the number of units of data contained in the file. The size of the unit is determined by the data characteristics of the file, discussed in section 2.2.2.

Archit size the maximum number of pages per archit. All the archits in this file, except the very last archit or those adjacent to holes, will be of this size.

Author ID the primary access group (see section 2.1.2) of the creator of the file.

Creation date the date and time of the creation of the file (set only once per file).

Data access date the date and time of the last access to the file (set when the file is opened).

Data modification date
the date and time of the last modification made to the data in the file (set when closing a file after opening it for modification, or when completing a transaction that has modified the file).

Header modification date
the date and time of the last modification to selected fields of the file header.

Advisory semaphore
for use by higher level mechanisms in local systems (see section 2.2.4). This is set to the primary access group id of the current holder of the semaphore.

Semaphore expiration time
the date and time that the semaphore will be automatically reset if it is not explicitly reset.

File print name a name specified when a file is created outside of any directory or the name under which the file is first entered into the directory hierarchy (see section 2.3.5).

Transaction count the number of modify transactions or closes after opens for modification which have been successfully completed on this file since its creation (see section 2.2.4).

Access control list a description of the access restrictions associated with the file (see section 2.2.3).

LFS information a block of data supplied by a local file system when storing files on the CFS (multiple local file systems may store information in this field only by common agreement as to the format and content).

File storage map physical storage information. This is implemented as a set of entries, one entry per archit. Each entry contains pointers to the corresponding data on secondary and archive storage. Archival information such as date of last access and date of last modification is also present in each entry. Appendix II describes the format of each entry.

2.3 Name Server

2.3.1 Overview

The Name Server provides three primary functions:

1. It maps names chosen by users to unique identifiers.[2]

2. It provides a directory structure that aids users in organizing their files in a logical manner.

3. It aids sharing of files by supporting shared directories and provides control over the extent of this sharing.

These services are provided completely independently of the services provided by the File Server. Consequently, it is possible to use the Name Server as a repository of names that have nothing to do with files. One such use, described in section 2.3.2, is to provide a network-wide naming scheme for IPC ports. Other uses of the Name Server are left to the imagination of the reader and the ingenuity of users.

The directory structure supported by the CFS is superficially similar to that in the UNIX file system [11]. This directory hierarchy can be viewed as a rooted graph with non-terminal nodes corresponding to directories and terminal nodes corresponding to names of entries in directories. An arbitrary path from the root to a leaf specifies the entry name at the leaf unambiguously. Similarly, an arbitrary path from the root to an internal node unambiguously specifies a directory. Such a specification of a file or directory name is called a *pathname* and is written as a '<' followed by the names, in order of traversal, of nodes encountered while traversing that path, the '>' character serving as a name separator. For convenience, the Name Server supports the notion of a *current directory*. This directory is specified by the user, and its pathname is logically prefixed to all pathnames specified without a leading '<' character.

2.3.2 Structure of a directory

Conceptually, a directory is a list of entries, each entry being a mapping from a *entry name* to a typed *entry value*. In the great majority of cases the *entry value* will be of type *FID*. This represents the simplest situation, where a name in a directory is directly mapped to the FID of a file. Since entries are merely mappings, it is perfectly possible and meaningful for a given file to have many names associated with it, in the same or different directories.

Subdirectories have entries of type *directory* in their parent directories. When parsing a pathname which contains a subdirectory name, the Name Server simply looks for the name of the subdirectory in the directory currently being searched and makes it the new search directory (provided of course that it is a directory). If there are many subdirectories in the pathname, this procedure is recursively repeated until an entry which is not of type *directory* is encountered. At this point, with the one exception described below, the corresponding entry value is returned as the result of the name lookup.

A second type of entry value is *Symbolic Link*. A symbolic link is the pathname of an entry in some

---

[2] In most cases this can be interpreted as "translates the name of a file to its FID."

directory and is used like a macro in a programming language. When the Name Server encounters a symbolic link while trying to resolve a name reference, it continues the search using the pathname corresponding to the symbolic link, thereby effectively performing a macro expansion. This may be applied recursively. To assist in locating errors, the Name Server provides a lookup function which does not perform this macro expansion of symbolic links.

Another type of entry value is *IPC port*. An IPC port may be associated with a name in a directory for two reasons. One use of this feature is to provide a simple name, IPC port mapping service so that users who wish to share IPC ports can remember names rather than IPC ports. Looking up an agreed-upon name in the CFS directory structure will yield an IPC port which can then be used for whatever purposes are necessary. Alternatively, an IPC port may be entered in a directory with the intention of simulating a file. All file access messages intended for the "file" will be rerouted by the File Server to the designated IPC port. The process controlling the port can then respond to these messages as appropriate to simulate file access.

To assist users who wish to use the Name Server for non-standard purposes, a special entry type called *UserDefined* is recognized by the Name Server. Entries of this type are integers whose values are neither checked nor interpreted by the Name Server. Entries of this type may be entered or looked up in directories just as entries of type *FID*. It is expected that higher-level mechanisms will check and interpret these entries suitably.

2.3.3 Versions

The naming mechanism supported by the Name Server reflects the expectation that most users will view individual files as being members of file families rather than as isolated entities. This is particularly true because the use of invariant files will typically lead to the creation of a sequence of files, closely related yet distinct from each other. As mentioned in section 2.2.2, editing an invariant file will result in a new invariant file; these two files are certainly distinct, yet they only represent two snapshots in the life history of what would be called a file in conventional file systems.

The Name Server captures this view by composing a full entry name from two parts: a *family name* and a *version number*. The family name is a text string that identifies a family of entry names. Version numbers are integers used to distinguish between members of a family. The version numbers of the members of a family reflect the chronological order in which the members were entered into the directory.

It is often the case that a user wishes to refer to the latest member of a family without bothering to remember or find out what its version number is. The Name Server therefore adopts the convention that a family name without a version number refers to the highest existing version. It also adopts the convention that a version number of 0 refers to the oldest existing version. Further, if no version number is specified when entering a name in a directory, the entry created will have its version number set to one larger than the

highest existing version of that family. If a version number is specified, it has to be larger than the highest existing version.

It should be borne in mind that the proper use of version numbers is up to the user. A malicious (or merely foolish) user may enter totally unrelated files as versions of a family. Or he may enter members of a family in an order that does not reflect their actual age --- version numbers will then be of dubious value.

2.3.4 Directory access rights

The current implementation of the directory structure will store directories through the File Server in stable storage as variant files. The Name Server uses the auxiliary access bits of a directory to define a set of directory access rights which it uses to control access to the directory.

These directory access rights are:

Lookup	the directory may be searched for names
CreateName	new names may be added to the directory
DeleteName	names may deleted from the directory (see section 2.3.6)
GetDirAccess	the directory access control list may be examined
SetDirAccess	the directory access control list may be modified
GetDefAccess	the default directory access control list may be examined
SetDefAccess	the default directory access control list may be modified.

Access checks on directories are performed by the Name Server in a manner analogous to that performed by the File Server on files, as described in section 2.2.3. Only the Name Server will have system rights such as *read* or *write* on a directory file.

Unlike conventional file systems, there is no notion of "ownership" of a directory. All access control is done via the access control list of the directory, and this list may be used to give a set of users equal rights on the directory. In some sense one can say that the "owners" of a directory are the access groups which have all rights on the directory. Thus, a set of users may collectively "own" a directory and share its use.

To assist in the sharing of files, every directory contains system and auxiliary *default access control lists*. The purpose of the default access control lists is to automatically modify the access control lists of files entered in that directory by users who have *SetSysAccess* or *SetAuxAccess* rights on the file. This modification takes place as follows:

- If no previous version of the name exists, the default access control lists are merged with the existing access control lists of the file.[3] For the system access control list to be modified the user must possess *SetSysAccess* rights; for the auxiliary access control list to be modified he must possess *SetAuxAccess* rights. If the user possesses only one of these rights, only the corresponding access control list is modified.

- If a previous version exists, the access control list from the previous version is merged with the existing access control list of the file. This is subject to the same access restrictions mentioned above.

A set of users having access to a common directory may set up the default access lists so that the mere entering of a name in that directory causes the corresponding file to be accessible to everyone in that set. If a user who does not have *SetSysAccess* or *SetAuxAccess* rights on a file enters it in a directory, the corresponding access control list of the file remains unchanged.

2.3.5 Creating new names

Files are created through the File Server independently of the directories in which they may reside. At the time that a name is entered into a directory, the Name Server checks to see if the name already exists in that directory. The behavior of the Name Server depends on whether a version number is specified as part of the name to be entered:

- *A version number is specified:*
  If the version number of the new name is less than or equal to that of the highest existing version, the entry is rejected. Otherwise the new entry name, entry value pair is entered in the directory.

- *No version number is specified:*
  A version number which is one larger than the highest existing version of the name is given to the new entry. If no previous version of the name exists, a version number of 1 is given to the new entry.

If the name maps to an entry value of type *FID*, the Name Server modifies the access control list of the corresponding file in the following way:

- *A previous version of the name exists:*
  The access control list for the most recent version is merged with that of the new version if the currently enabled access groups have appropriate rights on the file being entered. If the new version is to have more restricted access than the previous one, an explicit operation to restrict the access must be performed.

- *No previous version of the name exists:*
  The default access control list for the directory is merged with that of the file if the currently enabled access groups have appropriate rights on the file being entered. If the file being entered should have more restricted access than that specified by the default access control list, an explicit operation must be performed to restrict the access.

---

[3] In this context "merged" means that the resultant access control list is a union of the default access control list and the original access control list on the file.

If the name being entered is not of type *FID*, no access control list modification is attempted by the Name Server.

If the print name field of the file header is still unspecified when a file is first entered into a directory, it will be filled in with the entered name, provided the user has *WriteHeader* permission on the file.

The Name Server requires *CreateName* permission in the directory before a new name will be entered in a directory. For invariant files, the creator must also possess *Write* permission on the previous version, if one exists. This will be the only time that *Write* permission has any meaning for invariant files.

2.3.6 Deleting names

Since the CFS is intended to provide a permanent archive facility for all files, the traditional file deletion semantics is hardly appropriate. The standard CFS delete operation is simply a Name Server function which never deallocates any file storage. Thus, only entry names are deleted when it is necessary to clean up a directory. The file storage is simply allowed to migrate off-line through disuse. When a name is deleted from a directory, it is still retained in that directory, with a delete flag set. A subsequent undelete of the name causes the delete flag to be reset and it appears as if the name was never deleted at all. The actual removal of deleted names is accomplished by an *ExpungeName* operation on the directory. Once expunged, the corresponding entry name and entry value associations can only be recovered from the Name Archive (see section 2.3.7). However, the file itself is not removed from the file storage and may still be accessed via its FID.

2.3.7 Recovering expunged names

Corresponding to every directory is a Name Archive. This is a variant file to which only the Name Server has access rights. Each time a name is entered in a directory, it is also entered in the corresponding Name Archive, along with its entry value and the date and time of entry. The Name Archive is never deleted or truncated, even if the corresponding directory is. The Name Server supports a name retrieval service which can be used to recover the entry value that corresponded to a given pathname on a certain date and time. Since directories themselves have entries in their parent directories, this recovery mechanism will work even if some of the directories present in a pathname no longer exist. To use this retrieval service, a user must possess *LookUp* rights on the directory from which the pathname is to be retrieved.

2.4 Archive Server

The primary function of the Archive Server is to maintain, on tertiary storage, copies of all files ever created by the File Server on secondary storage. The design of the Archive Server is based on the premise that extremely large (of the order of 10^{10} bits per platter) random access write-once memories such as video disks

will be commercially available in the near future. The Archive Server will be run on a processor connected to the rest of the CFS via the network. Pending the availability of video disks, the Archive Server will use magnetic tapes and disks to simulate video disks.

The only component of the CFS that is aware of the presence of the Archive Server is the File Server. The relationship between secondary storage on the File Server and tertiary storage on the Archive Server is analogous to that between cache and main memory when a write-through caching strategy is used. Alterations to secondary storage cause tertiary storage to be updated. However, unlike a write-through cache, interactions between secondary and tertiary storage are asynchronous. There can be indeterminate delays between the queuing of a request by the File Server and its execution by the Archive Server. During this period the File Server may continue to attend to user processes and queue further requests to the Archive Server. On completion of a request, the Archive Server notifies the File Server of this fact.

The organization of information on tertiary storage is such that it is possible to recover every file that was successfully archived, even if all secondary storage is lost. This is achieved by archiving not only the contents but also the headers of files and by maintaining, in tertiary storage, a mapping table that maps FIDs to tertiary file header addresses.

For efficiency, however, pointers to data on tertiary storage are maintained in the file headers on secondary storage. Consequently, the File Server can request the Archive Server to retrieve data from specific addresses on tertiary storage --- no time-consuming FID-to-fileheader-to-data mappings have to be done by the Archive Server.

The unit of data transfer between secondary and tertiary storage is an archit. When a file is created by the File Server, a request to archive this file is spooled, awaiting attention by the Archive Server. Once archived, the File Server may erase any of the archits of the file from secondary storage with the assurance that it can retrieve that archit from the Archive Server at any time.

Invariant files need only be archived once, since they are never altered after creation. Variant files, however, have to be archived each time they are changed; i.e., each time a modify transaction is ended or a variant file open for modification is closed. Of course, only those archits that have actually been changed need to be archived.

The File Server uses a migration policy to erase from secondary storage those files which are not likely to be accessed for a "long" time. The definition of "long" is a policy decision based on parameters such as the demand for secondary storage, the characteristics of tertiary storage, and the long-term file access patterns. The migration algorithm uses the past access pattern of the file to migrate those files which are least likely to be used in the near future. To conserve secondary storage, file headers may also be migrated. However, archive retrieval requests can be much more efficient if the tertiary addresses of the relevant archits are

available on secondary storage. Consequently, the migration algorithm ensures that a file has been unused for a much longer period of time for its header to be migrated, than for its archits. Since archits of a file are independently migratable, it is quite possible for rarely-used archits of a file to be absent from secondary storage, while frequently-used archits are present. Accessing a migrated archit will automatically cause it to be retrieved from tertiary storage. The only indication that a user will have of this event is a real-time delay in the completion of his request to access data in that archit.

3. Primitive operations

3.1 Common characteristics

All requests of the CFS are made through IPC messages sent to ports of the File, Name or Authentication Servers. All request messages have a standard format. Each message consists of a message ID, a reply port, and one or more request parameters. The message ID is some value chosen by the user process. The CFS servers use this ID to identify the request when responding to messages. It will normally be updated by the user process for each new request so that sequential requests may be easily distinguished. The reply port specifies the destination for responses from the CFS. When a server responds to a request, the response message is sent to the specified reply port.

All requests have a *request code* which specifies the type of request being made. This is followed by a sequence of typed input parameters, which are interpreted according to the request code. Since each parameter is typed, the primitives support optional parameters.

Response messages from the CFS also have a common form. Each contains the source port from which the message was sent, the message ID of the request invoking the response, the completion code of the request, and one or more typed result values. The completion code specifies either that the request was successfully completed or that the request could not be completed successfully due to some error. When a completion code of *Success* is returned, the result values (if any) are the results of the completed operation. If an error occured, the completion code indicates the reason for the failure.

This request/reply message protocol is entirely asynchronous. User processes need not wait for a completion response from the CFS before proceeding. However, it is the responsibility of these processes to insure that no dependent actions are attempted before a completion response is received from the CFS.

3.2 Authentication Server primitives

3.2.1 General Overview

The Authentication Server primitives provide the verification mechanisms used in establishing connections with the CFS and manipulating access groups.

3.2.2 Primitives for establishing and breaking a connection

Logging in to the CFS

RequestCode: *Login*

Parameters: *user name* - a name which uniquely identifies each user. This is the same as the name of the primary access group to which the user belongs.

password - the password of the primary access group corresponding to the user.

Result: *File Server port* - an IPC port for communication with the File Server.

Name Server port - an IPC port for all communication with the Name Server.

Authentication Server port - an IPC port for all further communication with the Authentication Server.

Completion Code: *success* - Login successful

failure - User name not recognized or password incorrect

The *Login* request is used to establish a connection with the CFS. It verifies the authenticity of the user and creates and returns three ports for further communication: one to the File Server, one to the Name Server, and one to the Authentication Server. It associates the access group information from the GAF for the user with these ports so that it is available for access checks when necessary.

Logging off the CFS

RequestCode: *Logout*

Parameters: *none*

Result: *nothing*

Completion Code: *success* - Logout successful

The *Logout* request is used to break the connection previously established with the *Login* request. It is sent to the Authentication Server port created at login time and causes the connection to be closed down and the File, Name and Authentication Server ports to be deallocated. When the connection is broken all current open files are implicitly closed. Files which were open for transactions are closed with the abort flag set, others are closed normally.

3.2.3 Access group manipulation primitives

Changing a primary access group password

RequestCode: *ChangePassword*

Parameters: *user name* - the name of the primary access group whose password is to be altered.

old password - the old password for the access group.

new password - the new password for the access group.

Result: *nothing*

Completion Code: *success* - password changed

invalid password - old password incorrectly specified

invalid access group - specified access group does not exist

The *ChangePassword* request is used to to change the password for a primary access group. The old password must be supplied as a parameter before the new password may be changed. The System Administrator, however, need not supply the old password.

Enabling access groups

RequestCode: *EnableAccessGroup*

Parameters: *access group name* - the name of the access group to be enabled.

permanent flag - a flag indicating whether or not the access group should be permanently enabled.

Result: *nothing*

Completion Code: *success* - access group enabled

failure - user is not a member of the access group

The *EnableAccessGroup* request is used to activate an access group from the list of access groups to which the user belongs. Only the set of enabled access groups for a connection is used when checking access rights on a file. If the permanent flag is off, the access group is enabled only for the duration of the connection. If the permanent flag is on, the access group is enabled across logins, until explicitly disabled by the user.

Disabling an access group

RequestCode: *DisableAccessGroup*

Parameters: *access group name* - the name of the access group to be disabled.

permanent flag - a flag indicating whether or not the access group should be permanently disabled.

Result: *nothing*

Completion Code: *success* - access group disabled

failure - user is not a member of the access group

The *DisableAccessGroup* request is used to remove an access group from the protection environment of the requesting user. If the permanent flag is off, the access group is disabled only for the duration of the connection. If the permanent flag is on, the access group is disabled across logins, until explicitly enabled by the user.

Creating a new access group

RequestCode: *CreateGroup*

Parameters: *access group name* - the name to be given to this group for unique identification.

access group type - *primary* or *secondary*

full name (optional) - the full, descriptive name of the access group, used only for informational purposes

Result: *access group ID* - the FID of the GAF for the newly created access group.

Completion Code: *success* - new access group created

duplicate name - an access group whose name is *access group name* already exists.

access violation - only the System Administrator may create primary access groups

The *CreateGroup* request is used to create a new access group. The same primitive is used to create both primary and secondary access groups. However, only the System Administrator may create primary access groups. The access control list of the newly-created access group gives the creator all Authentication Server rights and gives no rights to anyone else. If the access group created is a primary access group, its password is set to null.

Deleting an access group

RequestCode: *DeleteGroup*

Parameters: *access group name* - the name of the access group to be deleted

Result: *nothing*

Completion Code: *success* - access group deleted

access violation - the user does not have *DeleteGroup* rights on this access group

nonexistent group - the specified access group does not exist.

The *DeleteGroup* request is used to delete an access group. Anyone possessing *DeleteGroup* rights on this access group is allowed to delete the group. Only the System Administrator may delete primary access groups.

Renaming an access group

RequestCode: *RenameGroup*

Parameters: *old name* - the current name of the access group

new name - the new name of the access group

Result: *nothing*

Completion Code: *success* - access group renamed

access violation - the user does not have *DeleteGroup* rights on this access group

duplicate name - the new name specified already exists

nonexistent group - the specified access group does not exist.

The user must possess *DeleteGroup* rights on an access group in order to rename it. The System Administrator may rename any access group.

Adding members to an access group

RequestCode: *AddToGroup*

Parameters: *access group*$_1$ - the name of the secondary access group to be modified.

access group$_2$ - the name of the access group to be added.

Result: *nothing*

Completion Code: *success* - *access group*$_2$ made a member of *access group*$_1$

access violation - the user does not possess *AddMember* rights on *access group*$_1$

redundant addition - *access group*$_2$ is already a member of *access group*$_1$

invalid access group - one of the specified access groups does not exist

The *AddToGroup* request is used to add a member to a secondary access group. The added member initially has *access group*$_1$ disabled in its GAF. A user must possess *AddMember* rights on an access group in order to add members to it.

Removing members from an access group

RequestCode: *RemoveFromGroup*

Parameters: *access group*$_1$ - the name of the secondary access group to be modified.

access group$_2$ - the name of the access group to be removed

Result: *nothing*

Completion Code: *success* - removal successful

access violation - the user does not possess *RemoveMember* rights on *access group*$_1$

not a member - *access group*$_2$ is not currently a member of *access group*$_1$

invalid access group - one of the specified access groups does not exist

The *RemoveFromGroup* request is used to remove members from a secondary access group. Removal of currently logged in members takes effect only when they log out or when they perform an action that causes their flattened access list to be recomputed.

Mapping Access Group Names to IDs

RequestCode: *TranslateGroupName*

Parameters: *access group name* - the name of the access group whose ID is desired

Result: *access group ID* - the ID of *access group name*

full name - the full name of the access group

Completion Code: *success* - the given name was successfully translated

invalid name - the given name does not belong to any access group

Mapping group IDs to names

RequestCode: *TranslateGroupID*

Parameters: *access group ID* - the ID of the access group whose name is desired

Result: *access group name* - the name of the access group whose ID was supplied

full name - the full name of the access group

Completion Code: *success* - the translation was successful

invalid ID - the given ID does not refer to an existing access group

Listing the members of a secondary access group

RequestCode: *ListMembers*

Parameters: *access group name* - name of the secondary access group whose members are to be listed

transitive closure flag - when this flag is set, both direct and indirect members of the access group are listed. Otherwise only direct members are listed.

list option flag - when set to *ID*, only access group IDs will be returned in *member list*; when set to *names*, only access group names will be returned; when set to *both*, a list of name, ID pairs will be returned

Result: *member list* - a list of of access groups which are members of this secondary access group. This may contain both primary and secondary access groups. The format of this result depends on *list option flag*

Completion Code: *success* - the members have been listed

nonexistent group - the secondary access group specified does not exist.

Listing the membership of an access group

RequestCode: *ListMembership*

Parameters: *access group name (optional)* - name of the primary or secondary access group whose membership in secondary access groups is to be listed. If none is specified, this parameter defaults to the current primary access group.

transitive closure flag - when this flag is set, both direct and direct membership of the access group is returned. Otherwise only direct membership is returned.

list option flag - when set to *ID*, only access group IDs will be returned in *membership list*; when set to *names*, only access group names will be returned; when set to *both*, a list of name, ID pairs will be returned

Result: *membership list* - the list of access groups of which *access group name* is a member. This list will contain only secondary access groups. Each entry in the list is marked *enabled* or *disabled*. The format of this result is determined by *list option flag*.

Completion Code: *success* - the membership list has been generated.

nonexistent group - the access group specified does not exist.

3.3 File Server primitives

3.3.1 General overview

The primitives provided by the File Server are used to directly modify file characteristics and contents. When a file is opened, an IPC port is returned by the File Server. This port is used in a manner similar to a file channel in conventional systems. Requests to read, write or close the file are sent to this port by the user. The file to be operated upon is thus implicitly identified by the port on which these requests are received. Other primitives require an explicit FID parameter to identify the file in question.

As was mentioned in section 2.3.2, it is possible to have IPC ports play the role of FIDs. File Server primitives will accept IPC port IDs wherever a FID is expected. All such requests will be forwarded to the process receiving on that port; replies from that process will go directly to the requesting user. The File Server merely plays the role of a middleman, making it appear to the user that he is operating on a file. In the descriptions of the primitives themselves, no mention is made of the use of IPC ports instead of FIDs --- keep in mind, however, that it is possible to use them and that the File Server treats them as described above.

Many of the primitives assume the existence of a *file input pointer* and a *file output pointer*. These are logical pointers into the file specifying where the next data unit is to be read from or where the next data unit is to be written.

3.3.2 File input and output primitives

Opening a file

RequestCode: *OpenFile*

Parameters: *FID* - the file ID of the file to be opened.

queue request flag - a flag which determines the action to be taken if the open fails due to the file being already open in a conflicting mode

mode - the mode in which the file is to be opened. This is some combination from the set *read, write, extend,* and *truncate.*

use transactions flag - if true, the file is opened for transactions in the mode specified by *mode.*

Result: *IPC port* - the port to which subsequent I/O request messages for the file should be sent.

Completion Code: *success* - the open was successful

request queued - the file was already open in a conflicting mode and *queue request* was *true.*

conflict - the file was already open in a conflicting mode and *queue request* was *false.*

access violation - the requesting user tried to open the file in a mode for which he does not possess rights.

The *OpenFile* request is used to initialize an existing file for reading and/or writing. If the open request is successful, an IPC port is generated and returned. This port is then used as the destination port for all subsequent I/O request messages for the file. The file input and output pointers are set to the first data unit of the file. The *mode* parameter must match the access rights granted to some currently enabled access group by the access control list of the file. The CFS checks to see if the file is already open in a conflicting mode, as described in section 2.2.4. If a conflict exists, the action taken depends on *queue request*: when true, the request is queued and *success* will be sent by the CFS to the user when the open succeeds (this message is in addition to the reply message sent immediately). Until then the *IPCport* cannot be used for any file operation except *CloseFile.*

Reading file data

RequestCode: *ReadFile*

Parameters: *data unit count* - the number of data units to read from the file starting at the indicated position.

position (optional) - the data unit position in the file at which to begin the read. If no position is specified, the file input pointer is used.

Result: *data* - the data units read from the file. This will never be more than the amount of data requested by the data unit count but may be less if the read began or extended past the end of the file or if a hole was encountered.

data length - the actual number of data units read; this will be equal to *data unit count* unless a hole or end of file is encountered.

hole length - the length of the hole skipped after reading data.

Completion Code: *success* - read was successful

EOF - the file was positioned at the end of the file at the start of the read

nonexistent data - the file was positioned at or within a hole at the start of the read.

file not open - the file is not yet open

mode violation - this file has not been opened for *read*

transaction violation - the file is open for transactions but the data to be read is not within the pages reserved by this transaction

The *ReadFile* request reads the file associated with the port receiving the request and returns the requested data units. In the most general case, a read consists of reading data followed by a hole. The file input pointer is updated to point to the data unit following the last data unit read, or the end of the hole if one was encountered. The port must have been opened for *read* when it was created. If the file was opened for transactions, the read may only be performed within a transaction. The EOF and nonexistent data completion codes are returned only if the file was positioned at the end of a file or within a hole at the beginning of the read operation. In other cases the user must use *data length* and *hole length* to check whether a hole or the end of file was encountered during the read.

Writing a file

RequestCode: *WriteFile*

Parameters: *data* - the data to be written to the file starting at the indicated position.

position (optional) - the position (expressed in number of data units) in the file at which to begin the write. If no position is specified, the file output pointer is used.

Result: *nothing*

Completion Code: *success* - the data was successfully written

file not open - this file is not yet open

mode violation - the file is not open for modification, or it was not open for *extend* and the write would have extended the file.

transaction violation - the file is open for transactions and some of the data written would have been on pages not reserved for this transaction.

The *WriteFile* request writes data to the file associated with the port receiving the request. The file output pointer is updated to point to the data unit following the last data unit written. The port must have been opened for *write* to overwrite data which currently exists and for *extend* to write new data past the current end of file. If the file was opened for transactions, this operation may only be performed within a transaction.

Positioning file pointers

RequestCode: *PositionFile*

Parameters: *input offset* - the signed offset, expressed in data units at which to position the file input pointer. The offset is expressed with respect to a base defined by the next parameter.

input offset base - specifies the base with respect to which the input offset is specified. A value of *start* signifies that the offset is with respect to the beginning of the file, *current* signifies that is is with respect to the current position, and *end* signifies that it is with respect to the end of the file.

output offset - the offset, expressed in data units, at which to position the file output pointer. The offset is expressed with respect to a base defined by the next parameter.

output offset base - similar to *input offset base*

Result: *new input pointer* - absolute value of new position of input pointer.

new output pointer - absolute value of new position of output pointer.

Completion Code: *success* - the resultant pointers are within the current bounds of the file. The output pointer may be set to a point beyond the end of the file provided the file has been opened for *extend.*

failure - one or both pointers were set to a value outside the current bounds of the file. This code is also returned if the file was not open for *read* and an attempt was made to set the input pointer, or if the file was not open for *write* or *extend* and an attempt was made to set the output pointer. The pointers are not altered.

file not open - the file is not yet open

The *PositionFile* request updates the file I/O pointers for the file associated with the port receiving the request. The port must have been opened for *write*, or *extend* to position the output pointer, and *read* to position the input pointer. The resultant input pointer should always be within the current bounds of the file. If the file is opened for *extend* the output pointer may be set to a position beyond the end of the file; otherwise it too must be set to a position within the current limits of the file.

Truncating a file

RequestCode: *TruncateFile*

Parameters: *new length (optional)* - the new size of the file, in data units. If not specified the current position of the file output pointer will be taken as the new end of the file.

Result: *nothing*

Completion Code: *success* - the file has been truncated to the desired size.

invalid limit - the new value specified is beyond the current end of the file.

access violation - the file is not open for *truncate*

file not open - the file is not yet open

The *TruncateFile* primitive is used to truncate a file. The file must have been opened to permit *truncate*. The new size of the file may be explicitly specified or inferred by the File Server to be the current output pointer position. The resulting size of the file should be less than or equal to the current size.

Specifying a transaction

RequestCode: *StartTrans*

Parameters: *page list* - the list of *page range, mode* pairs to be in the transaction. Each page range will be set to permit operations of the type specified in the corresponding mode.

queue request flag - a flag which determines the action to be taken by the CFS when one or more of the pages requested is already in use, in a conflicting mode.

Result: *transaction ID* - the ID of a newly created transaction.

Completion Code: *success* - transaction was initiated.

request queued - some of the specified pages were being used by another transaction in a conflicting mode; this request has been queued.

conflict - some of the specified pages were being used by another transaction in a conflicting mode; this request has not been queued.

mode violation - the file was not opened to permit one of the modes specified in *page list*.

file not open - the file is not yet open

The *StartTrans* request is used to initiate a transaction on the file associated with the port receiving the request. *Page list* is a list of pairs of the form $(p_1{:}q_1, m_1), (p_2{:}q_2, m_2), \ldots$ where the p's and q's represent the starting and ending page numbers of a page range and the m's represent the mode in which the corresponding page range will be used. Ranges must not overlap, each p must be less than or equal to the corresponding q, and all the p's and q's must be less than maximum number of pages of the file, unless the corresponding m contains *extend*. A value of *END* for a q means "the rest of the file". This is particularly useful for *extend* transactions. No other currently existing transaction on the same file may overlap any of the pages in *page list* in a conflicting mode. The action taken by the CFS in case such a conflict exists, depends on *queue request*. If true, the request is queued. When the pages are available a message indicating *success* is sent to this user; this message is in addition to the reply message sent immediately.

Augmenting a transaction

RequestCode: *AugmentTrans*

Parameters: *page list* - the list of *page range, mode* pairs to be added to the transaction. Each page range will be added to permit operations of the type specified in the corresponding mode.

queue request flag - a flag which determines the action to be taken by the CFS when one or more of the pages requested is already in use, in a conflicting mode.

transaction ID - the ID of the transaction to be augmented

Result: *nothing*

Completion Code: *success* - specified pages were added to the transaction in the requested modes

request queued - some of the specified pages were being used by another transaction in a conflicting mode; this request has been queued.

conflict - some of the specified pages were being used by another transaction in a conflicting mode; this request has not been queued.

mode violation - the file was not opened to permit one of the modes specified in *page list.*

nonexistent transaction - the specified transaction ID is invalid.

file not open - the file is not yet open

The *AugmentTrans* request is used to add pages to a transaction on the file associated with the port receiving the request. *Page list* is a list of pairs of the form $(p_1{:}q_1, m_1), (p_2{:}q_2, m_2), \ldots$ where the p's and q's represent the starting and ending page numbers of a page range and the m's represent the mode in which the corresponding page range will be used. Ranges must not overlap, each p must be less than the corresponding q, and all the p's and q's must be less than maximum number of pages of the file, unless the corresponding m contains *extend.* No other currently existing transaction on the same file may overlap any of the pages in *page list* in a conflicting mode. The action taken by the CFS in case such a conflict exists, depends on *queue request.* If true, the request is queued. When the pages are available, they are marked as being in use by this user and a reply indicating their availability is sent to this user. This is in addition to the reply message sent immediately.

Terminating a transaction

RequestCode: *EndTrans*

Parameters: *transaction ID* - the transaction ID of the transaction to be completed.

abort flag - a flag indicating whether or not the changes made within the transaction should be aborted.

Result: *nothing*

Completion Code: *success* - the specified transaction was successfully ended

nonexistent transaction - the specified transaction ID is invalid.

The *EndTrans* request is used to complete a transaction on the file associated with the port receiving the request. The transaction to be terminated is indicated by the transaction ID. If the abort flag is off, and if the file was modified, the transaction count of the file is incremented and all of the changes made within the transaction are indivisibly performed on the file. If the abort flag is on, then the transaction is aborted and none of the changes made within the transaction are actually made on the file.

Closing a file

RequestCode: *CloseFile*

Parameters: *abort flag* - a flag indicating whether or .not the I/O associated with the file should be aborted.

Result: *file modification date* - indicating the modification date stored by the CFS in the file header.

Completion Code: *success* - always

The *CloseFile* request is used to release the open file channel associated with the port receiving the request and deallocate the port. It implicitly completes all uncompleted transactions active for the file port. If the file was in the process of being created, it is made available for access through the File Server. When the abort flag is on, any uncompleted transactions for the port are instead aborted without any changes to the file and, if the file was being newly created, it is deallocated and discarded without becoming part of the CFS. The file modification date is returned primarily for use by local file systems, which require it to ensure consistency of local copies of files.

Creating a new file

RequestCode: *CreateFile*

Parameters: *storage characteristics* - the class of storage in which the file should be recorded. This is either *standard* if the file should be stored in standard storage or *stable* if the file should be stored in stable storage.

data characteristics - one of *universal text, 8-bit binary, 16-bit binary, 32-bit binary* or *36-bit binary.*

mutability - the file class to which the file belongs. This is either *variant* or *invariant.*

advisory file type (optional) - the file type to be assigned to the file.

print name (optional) - a name to be recorded for the file in its file header. This name may also be set by the Name Server when the file is entered into a directory if no name has yet been specified. When the file will never be entered in any directory, this name may be useful in recording its identity.

archit size (optional) - the size of the archival unit expressed in pages. If absent, a suitable default is chosen by the File Server.

Result: *IPC port* - a port opened for *read*, *write*, *truncate* and *extend* to which all subsequent file I/O requests should be sent.

FID - the file ID of the newly created file. This FID may be stored by the user process but will not be recognized by the File Server until the file port is closed.

Completion Code: *success* - always

The *CreateFile* request is used to create a file on the CFS. It allocates a file header and file ID for a new file and initializes the header as follows. The file properties and print name are set according to the parameters. The file size and transaction count are zeroed. The advisory semaphore information is cleared and the local file system block and storage map are initialized empty. The access, creation and modification dates are set to the current date and time. The author ID is filled in with the user ID and an access control list is supplied which grants all permissions to the primary access group and none to anyone else. An IPC port for writing the file is generated and returned. If a file is an invariant file, this is the only time that it will be possible to write data to it.

3.3.3 File header manipulation primitives

Reading the file header

RequestCode: *GetHeader*

Parameters: *FID* - the file ID of the header to be read

field mask - a bit vector describing the fields of the header to be returned. Bit n corresponds to field n of the header (see appendix I). If the corresponding bit of the field mask is on, that field of the header is returned.

Result: *header fields* - each field of the file header specified by the field mask is returned. The fields are returned in numeric order and each is typed as to its contents.

Completion Code: *success* - requested fields have been returned

invalid FID - the FID specified does not refer to an existing file

access violation - the enabled access groups do not possess access rights to read the file header.

The *GetHeader* request reads the file header and returns the specified fields. It requires *ReadHeader* permission on the file.

Changing the file header

RequestCode: *SetHeader*

Parameters: *FID* - the file ID of the header to be changed.

field mask - a bit vector specifying the fields of the header to be changed (as in the *GetHeader* request).

fields - the new values for the specified fields of the header.

Result: *header modification date* - the date entered by the File Server into the corresponding file header field. This value is used by local file systems for synchronizing local copies of file headers.

Completion Code: *success* - the requested fields have been modified

invalid FID - the specified FID does not refer to an existing file.

access violation - the enabled access groups do not possess rights to modify the file header

The *SetHeader* request is used to update particular fields of the file header. It requires *WriteHeader* access permission in the access control list of the file. Only certain fields of the file header may be changed with this request (see appendix I).

Reading the access control list

RequestCode: *GetAccess*

Parameters: *FID* - the file ID of the access control list to be read.

sys/aux flag - if set to *sys* the system access control list for the file is retrieved; if set to *aux* the auxiliary access control list is retrieved; *both* causes both lists to be retrieved

Result: *access control list* - the access control list of the file. If *sys/aux flag* is set to *sys* or *aux*, a list of two-tuples consisting of access group ID, access rights pairs is returned. If the *sys/aux flag* specifies both lists, a list of three-tuples consisting of access group ID, system rights and auxiliary rights is returned.

Completion Code: *success* - the access control list was successfully read.

invalid FID - the FID specified does not refer to an existing file

access violation - the enabled access groups do not possess rights to read the specified access control list of this file. *GetSysAccess* rights are needed to read the system access control list and *GetAuxAccess* rights are needed to read the auxiliary access control list.

Changing the access control list

RequestCode: *SetAccess*

Parameters: *FID* - the file ID of the access control list to be updated.

sys/aux flag - if set to *sys* the system access control list is modified; if set to *aux* the auxiliary access control list is modified; *both* causes both access control lists to be modified

rights list - the new access rights for the corresponding access group(s). If *sys/aux flag* is set to *sys* or *aux*, this consists of a list of access group, access rights pairs. If the flag is set to specify both system and auxiliary rights, this list consists of a set of three-tuples corresponding to access group IDs, system rights and auxiliary rights.

Result: *nothing*

Completion Code: *success* - the access control list was successfully modified

invalid FID - the FID specified does not refer to an existing file.

access violation - the user does not possess sufficient rights to modify the specified access control list of this file. *SetSysAccess* rights are needed to set the system access control list and *SetAuxAccess* rights are needed to set the auxiliary access control list.

The *SetAccess* request is used to update the access rights for one or more access groups in the access control list of a file. If the access rights are null, the access group is completely removed from the access control list of the file. If the access rights are non-null and the access group is already on the access control list, the new rights are substituted for the old rights. If the access rights are non-null and the access group is not currently on the access control list, the access group is added to the access control list with the specified access rights.

Checking access on a file

RequestCode: *CheckAccess*

Parameters: *FID* - the FID of the file to be checked

access group list (optional) - the list of IDs of the access groups whose rights are to merged for the purposes of this check. If absent, the set of currently enabled access groups is used.

Result: *rights mask* - a bit mask corresponding to system and auxiliary rights.

Completion Code: *success* - the rights list for the specified access groups on the file have been returned

invalid FID - the specified FID does not belong to an existing file

access violation - the set of currently enabled access groups does not have *GetAccess* privileges on this file

The *CheckAccess* primitive is used to check the access that any arbitrary collection of access groups have on a file. The *GetAccess* privilege is needed to perform this check unless the access groups being checked are the currently enabled ones.

3.3.4 Miscellaneous primitives

Setting the Advisory Semaphore

RequestCode: *SetSem*

Parameters: *FID* - the ID of the file whose semaphore is to be set

timeout period - the length of time after which the CFS should automatically reset the semaphore if it is not explicitly reset.

Result: *current holder* - the ID of the user who has the semaphore set.

expiration time - the time at which the semaphore will time out.

Completion Code: *success* - the semaphore was originally clear and was set by this call

failure - the semaphore was already set.

invalid FID - the FID specified does not refer to an existing file.

access violation - the currently enabled access groups do not have *read* access on the file.

The Advisory Semaphore on the file is set. If the semaphore is already set, the semaphore expiration time and the ID of the user who currently has it set are returned. The semaphore will automatically be reset if it is not explicitly reset within the time period specified. The user must possess *read* rights to the file.

Resetting the Advisory Semaphore

RequestCode: *ResetSem*

Parameters: *FID* - the ID of the file whose Advisory Semaphore is to be reset.

Result: *nothing*

Completion Code: *success* - the semaphore has been reset

failure - the requesting user did not have the semaphore set

invalid FID - the specified FID does not refer to an existing file.

access violation - the currently enabled access groups do not have *read* rights on the file

The *ResetSem* request is used to reset the semaphore associated with the specified file. Only the user who set the semaphore may reset it.

Purging file storage

RequestCode: *PurgeFile*

Parameters: *FID* - the file ID of the file to be purged.

Result: *nothing*

Completion Code: *success* - the file was successfully purged

invalid FID - the FID specified does not refer to an existing file

access violation - the enabled access groups do not possess the right to purge the file.

The *PurgeFile* request is used to completely purge a file from the CFS file storage facility. It requires *purge* permission in the access control list of the file. This request completely destroys the file and deallocates its file header and storage map. It is intended primarily for use in destroying sensitive files and should rarely be used. Once purged, a file cannot be recovered even from archive. If a file is purged while users have it open, an abort will be sent to the open file ports.

3.4 Name Server primitives

3.4.1 General overview

The Name Server provides primitives to map names to their entry values, to create and delete directories, and to control access to directories.

The directory structure has already been described in detail earlier. To recapitulate, a pathname consists of a sequence of zero or more directory names separated by '>' characters followed by a terminal name. The pathname is interpreted to begin at the root of the directory hierarchy if it begins with a '<' character, and from the current directory otherwise. The version number of a name consists of a ';' character followed by some sequence of digits. The version number defaults to the highest existing version number when none is

explicitly specified in the name. A version number of 0 is taken to mean the oldest existing version. Note that this defaulting of version is done by only three primitives: *LookUpName*, *EnterName* and *CreateDir*. All other primitives require explicit version numbers. A pathname which terminates in a non-leaf node identifies a directory.

Entry names accepted by the Name Server may be of arbitrary length and may contain any printable ASCII characters other than '*', '<', '>', ';' and '?'. Two primitives, *ScanNames* and *ScanArchivedNames* permit wildcarding in the terminal component of the pathnames that they accept as input. Two forms of wildcarding are recognized. A '?' character in a name will match exactly one arbitrary ASCII character. A '*' represents an arbitrary number of ASCII characters. Both forms of wildcarding may be used within one name.

3.4.2 Name manipulation primitives

Looking up names

RequestCode: *LookUpName*

Parameters: *pathname* - the name to be looked up (may not contain wildcard characters)

Result: *entry value* - the value and the full pathname of the corresponding entry.

Completion Code: *success* - the name was successfully looked up.

not found - the specified name was not found.

directory - *pathname* refers to a directory; *Result* is undefined.

access violation - the user does not have *Lookup* rights on the directory in which the lookup was attempted

This function provides a simple way to look up a name. If any entry value encountered during the parsing of *pathname* is of type *Symbolic Link*, a macro expansion is performed using the value of that entry in place of the corresponding name in *pathname*. If the final result is of type *FID* or *IPC Port*, the corresponding FID or IPC port is returned. If it is of type *Directory* no result is returned, but *Completion Code* specifies the fact that a lookup on a directory name was attempted. If *pathname* does not contain a version number, the most recent version is assumed. If a version number of 0 is specified, the oldest version is assumed.

Scanning names

RequestCode: *ScanNames*

Parameters: *pathname* - the pathname to be looked up (may contain wildcard characters in the terminal component).

name status - whether active names, deleted names or both are to be considered

previous name (optional) - the final name returned from the last call to *ScanNames* (see explanation below).

name count (optional) - the maximum number of name, entry value pairs to return.

Result: *entry value list* - the fully expanded names and entry values of the next set of names in the directory hierarchy which match *pathname.*

actual name count - the actual number of name, entry value pairs returned.

Completion Code: *success* - the given pathname was successfully scanned

access violation - the user does not possess *LookUp* rights on the directory to be scanned

illegal pathname - wildcards were found in *pathname* at other than the terminal component

The *ScanNames* request is used to map name expressions to entry values. This primitive differs from mere iteration on *LookUpName* in that the latter expands *Symbolic Links* in the terminal component of *pathname* whereas *ScanNames* does not. The directory hierarchy is searched for file names matching the given name. Depending on the value of *name status*, only active names, deleted names or both are considered. If more than one name matches because wildcard characters were included in the name then all are returned up to the limit specified in the request. If no limit is specified, infinity is assumed. The actual number of names listed is returned in *actual name count. ScanNames* behaves as a generator of names matching the specifications. If *previous name* is specified, generation continues after that name. Otherwise generation begins at the very first matching name. If no version number is specified, then the highest existing version of the name is returned. If the version number is wildcarded, then all existing versions of the name are returned. Note that wildcarding is permitted **only** in the terminal component of *pathname.* Names corresponding to directories will be returned with type *directory* but will have undefined values.

Entering a name in a directory

RequestCode: *EnterName*

Parameters: *pathname* - the name to be entered into the directory structure (no wildcards).

FID - the file ID to be entered with the name

or

IPC port - the IPC port to be entered with the name

or

symbolic link - the symbolic link to be entered.

or

UserDefined type value - an arbitrary integer value

or

directory name - the name of a directory to be entered

Result: *new name* - the full pathname, including version number, actually entered.

Completion Code: *success* - *pathname* was successfully entered

conflicting version - the version number specified in *pathname* is less than or equal to that of an already existing version

access violation - the user does not possess *CreateName* rights in the directory in which the name is to be entered (i.e., in the penultimate component of *pathname*). If a version of the same name already exists and corresponds to an invariant file, the user must also possess *Write* privileges on the file.

The *EnterName* request is used to place a name and entry value pair in the directory structure. It requires *CreateName* permission in the directory within which the name is being entered. If an existing version of the name corresponds to an invariant file, *Write* privileges on the file are needed. If a previous version of the name already exists in the directory and no version number is specified in *pathname*, the name is entered with the next higher version number. If a previous version of the name already exists and a version number is specified in the name, then it must be greater than the highest version number so far. If no previous version of the name has ever existed, then version one is assigned if none is specified in *pathname*, otherwise the specified version is used.

Removing a name from a directory

RequestCode: *DeleteName*

Parameters: *pathname* - the name to be deleted.

Result: *nothing*

Completion Code: *success* - the name was successfully deleted

nonexistent name - the name specified does not exist

access violation - the user does not possess *DeleteName* rights on the directory from which the name is to be deleted.

The *DeleteName* request is used to delete a name from a directory. It requires *DeleteName* permission in the directory. The name is marked as deleted, but remains in the directory.

Restoring a deleted name

RequestCode: *UndeleteName*

Parameters: *pathname* - the name to be undeleted.

Result: *nothing*

Completion Code: *success* - the name was successfully deleted

nonexistent name - the name specified does not exist

access violation - the user does not possess *DeleteName* rights on the directory from which the name is to be restored.

The *UndeleteName* request is used to restore a deleted name. It requires *DeleteName* permission on the directory in which the entry resides.

Expunging deleted names

RequestCode: *ExpungeNames*

Parameters: *pathname* - name of the directory from which files are to be expunged.

entry names (optional) - the names of the entries to be expunged from *pathname.*

Result: *nothing*

Completion Code: *success* - the expunge was successful

not deleted - some of the names specified were not deleted names in the directory. No expunging is done.

access violation - the user does not possess *DeleteName* rights on the directory from which the name is to be deleted.

The specified names are expunged from the directory. If no names are specified, all deleted names are expunged. The only way to recover the expunged name-FID associations is via the Name Archive. The user must possess *DeleteName* permission in the directory for this operation to succeed.

Retrieving archived names

RequestCode: *ScanArchivedNames*

Parameters: *pathname* - the name to be scanned for; may be wildcarded

start time - a date and time which determines the maximum age of the names scanned;

here "age" is determined by the date of entry in the directory whose name archive is being scanned

end time - a date and time which determines the minimum age of the names scanned; here "age" is determined by the date of entry in the directory whose name archive is being scanned

name count (optional) - the maximum number of name, entry value pairs to be returned. If this parameter is not specified, infinity is assumed.

Result: *entry value list* - a list of the names, entry values and creation entry dates corresponding to *pathname* between *start time* and *end time*

actual name count - the actual number of name, entry value pairs returned.

Completion Code: *success* - the entry values were successfully retrieved

access violation - the user does not possess *Lookup* rights the directory whose name archive is being scanned

Retrieves the entry values corresponding to the given name between the specified dates and times. The user must possess *Lookup* rights on the directory whose name archive is begin scanned.

3.4.3 Directory manipulation primitives

Creating a directory

RequestCode: *CreateDir*

Parameters: *pathname* - the name of the directory to be created.

Result: *nothing*

Completion Code: *success* - the directory was created

access violation - the enabled access groups do not have *CreateDir* permission in the parent of the directory to be created.

invalid name - one of the directories in *pathname* does not exist.

conflicting version - the version number specified in *pathname* is less than or equal to that of an existing version of the name.

The *CreateDir* request is used to add new directories to the directory hierarchy. It requires *CreateName* permission in the parent of the directory to be created. The directory is created with the access control list and default access control list both copied from the parent directory. In addition, the creator is given *SetDirAccess* rights on the directory. An empty Name Archive is also created for this directory.

Reading the directory access control list

RequestCode: *GetDirAccess*

Parameters: *directory name* - the name of the directory whose access control list is to be read.

access control list - the access control list of the directory.

Result: *nothing*

Completion Code: *success* - the access control list is returned

invalid name - directory name does not refer to a directory, or one of the directories specified in the pathname is nonexistent.

access violation - the enabled access groups do not have *GetDirAccess* rights on the directory.

The *GetDirAccess* request is used to retrieve the access control list associated with a directory. It requires *GetDirAccess* permission in the directory whose access control list is being read.

Changing the directory access control list

RequestCode: *SetDirAccess*

Parameters: *directory name* - the name of the directory whose access control list is to be updated.

rights list - the list of access ID, access rights pairs to be added to the access control list.

Result: *nothing*

Completion Code: *success* - the access control list was successfully modified

invalid name - directory name either does not exist or does not refer to a directory

access violation - the user does not possess *SetDirAccess* on the directory

The *SetDirAccess* request is used to update the access rights for one or more access groups in the access control list of a directory. It requires *SetDirAccess* permission in the directory whose access control list is being changed. If the access rights are null, the access group is completely removed from the access control list of the file. If the access rights are non-null and the access group is already on the access control list, the new rights are substituted for the old rights. If the access rights are non-null and the access group is not currently on the access control list, the access group is added to the access control list with the specified access rights. The directory access rights are specified in section 2.3.4.

Checking access to a directory

RequestCode: *CheckDirAccess*

Parameters: *directory name* - the name of the directory to be checked

access group list (optional) - the list of IDs of the access groups whose rights are to merged for the purposes of this check. If absent, the set of currently enabled access groups is used.

Result: *rights mask* - a bit mask specifying some subset of the directory access rights

Completion Code: *success* - the rights list for the specified access groups on the directory has been returned

invalid directory name - the specified name does not belong to an existing directory

access violation - the set of currently enabled access groups does not have *GetDirAccess* privileges on this directory

The *CheckDirAccess* primitive is used to check the access that any arbitrary collection of access groups have on a directory. The *GetDirAccess* privilege is needed to perform this check.

Reading the default access control list

RequestCode: *GetDefAccess*

Parameters: *directory name* - the name of the directory whose default access control list is to be read.

sys/aux flag - if set to *sys* the default system access control list is retrieved; if set to *aux* the default auxiliary access control list is retrieved; *both* causes both lists to be retrieved.

Result: *access control list* - the default access control list of the directory. If the *sys/aux flag* is set to *sys* or *aux*, this consists of a list of access group, access rights pairs. If the flag is set to *both*, this list consists of a set of three-tuples corresponding to access group IDs, system rights and auxiliary rights.

Completion Code: *success* - the access control list was successfully returned

invalid name - *directory name* either does not exist or does not refer to a directory

access violation - the enabled access groups do not possess *Lookup* on the directory

The *GetDefAccess* request is used to retrieve the default access control list associated with a directory. It requires *GetDefAccess* permission in the directory whose default access control list is being read.

Changing the directory access control list

RequestCode: *SetDefAccess*

Parameters: *directory name* - the name of the directory whose default access control list is to be updated.

sys/aux flag - if set to *sys* the default system access control list is modified; if set to *aux* the default auxiliary access control list is modified; *both* causes both lists to be modified.

rights list - the new access rights for the corresponding access group(s). If the *sys/aux flag* is set to *sys* or *aux*, this consists of a list of access group, access rights pairs. If the flag is set to *both*, this list consists of a set of three-tuples corresponding to access group IDs, system rights and auxiliary rights.

Result: *nothing*

Completion Code: *success* - the access control list was successfully modified

invalid name - *directory name* either does not exist or does not refer to a directory

access violation - the enabled access groups do not possess *SetDefAccess* rights on the directory

The *SetDefAccess* request is used to update the access rights for one or more access groups in the default access control list of a directory. It requires *SetDefAccess* permission in the directory whose default access control list is being changed. If the access rights are zero, the access group is completely removed from the default access control list of the directory. If the access rights are non-zero and the access group is already on the access control list, the new rights are substituted for the old rights. If the access rights are non-zero and the access group is not currently on the access control list, the access group is added to the access control list with the specified access rights.

Checking default access on a directory

RequestCode: *CheckDefAccess*

Parameters: *directory name* - the name of the directory to be checked

access group list (optional) - the list of IDs of the access groups whose rights are to merged for the purposes of this check. If absent, the set of currently enabled access groups is used.

Result: *rights mask* - a bit mask specifying some subset of the file access rights

Completion Code: *success* - the default rights list for the specified access groups on the directory have been returned

invalid directory - the specified name does not belong to an existing directory

access violation - the set of currently enabled access groups does not have *GetDefAccess* privileges on this directory

The *CheckDefAccess* primitive is used to check the default access that any arbitrary collection of access groups has on a directory. The *GetDefAccess* privilege is needed to perform this check.

Changing the current directory

RequestCode: *ChangeDir*

Parameters: *directory name* - the name of the directory to be used as the new current directory for all names searches.

Result: *nothing*

Completion Code: *success*

The *ChangeDir* request is used to change the name of the directory used as the current directory for the connection.

Acknowledgements

The authors wish to acknowledge the contributions of Rick Rashid and Gene Ball, who took part in a number of the design meetings and critiqued the final design. They also wish to express their appreciation to the members of the Department of Computer Science, in general, and the members of the Spice group, in particular, for their comments on the design and on this document.

I. File header fields

The following table specifies the fields of the file header and indicates which of them may be explicitly or implicitly altered. The four primitives *SetHeader, SetAccess, CreateFile, SetSem* and *ResetSem* allow a user to explicity alter certain fields. Certain other fields are implicitly set by the primitives *CreateFile, OpenFile, CloseFile, EndTrans, SetHeader, SetAccess* and *EnterName.*

	SetHeader	*SetAccess*	*CreateFile*	*Set/Reset Sem*	Implicit
File ID					1
File Properties:					
Mutability			X		
Storage Characteristics			X		
Data Characteristics			X		
Advisory File Type	X		X		
File Size					1, 3, 4
Archit Size			X		1
Author ID					1
Creation Date					1
Data Access Date					1, 2
Data Mod. Date					1, 3, 4
Header Mod. Date					1,3,4,6,7
Advisory Semaphore				X	
Sem. Expiration Time				X	
File print name	X		X		5
Transaction count					1, 4
Access control list		X			1, 5
Local file system info.	X				
Storage Map					1. 3, 4

Legend for Implicit

1. *CreateFile*
2. *OpenFile*
3. *CloseFile*
4. *EndTrans*
5. *EnterName*
6. *SetHeader*
7. *SetAccess*

II. Storage Map Format

The storage map consists of a list of *archit descriptors.* Each such descriptor contains the following fields:

- *Starting page number*
- *Length of archit in pages*
- *Date and time archit was last modified*
- *Date and time archit was last accessed*
- *Date and time archit was last archived*
- *Pointer to secondary storage copy of archit:* the pointer is typed to indicate whether it points to a contiguous set of physical pages or to a non-contiguous set of physical pages via a page table.
- *Pointer to tertiary storage copy of archit:* the pointer is typed to indicate whether it points to a contiguous set of physical pages or to a non-contiguous set of physical pages via a page table.

The file header itself is the very first archit in the file; archival information about the header is found in this archit.

III. Page Size Table

Data Type	No. of Data Units per page
8-bit binary	512
16-bit binary	256
32-bit binary	128
36-bit binary	102
Universal text	??

IV. Format of the GAF

For every access group in the system there is one GAF file containing the following information:

Access Group Name	a unique name that identifies this access group. For primary access groups this is the login name of the corresponding user.
Full Name	the full descriptive name of the access group. For primary access groups this will be the full name of the corresponding user; consequently it may not be unique.
Access Group ID	a unique integer that identifies this access group. Used instead of *Access Group Name* wherever a fixed length identification is needed for the access group, such as in the access control list of files. The FID of the GAF is used as the access group ID in the current implementation.
Group Type	(Primary/Secondary)
Password	For primary access groups this is the login password. Not present for secondary access groups.
Direct Members List	the list of direct (i.e., without applying transitive closure) members of this access group
Direct Membership List	the list of access groups of which this access group is a direct member; the entry for each such access group in this list is flagged *Enabled* or *Disabled*

V. Summary of Primitives

The following is a summary of the primitives provided by the CFS. Optional parameters are underlined. Alternatives for a given parameter are enclosed within curly brackets; only one of these alternatives may be selected. The page on which the primitive is fully described appears within square brackets.

Authentication Server Primitives

[22] ***Login*** *(user name, password) ⇒ (File Server port, Name Server port, Authentication Server port)*

[22] ***Logout*** *( ) ⇒ ( )*

[23] ***ChangePassword*** *(user name, old password, new password) ⇒ ( )*

[23] ***EnableAccessGroup*** *(access group name, permanent flag) ⇒ ( )*

[23] ***DisableAccessGroup*** *(access group name, permanent flag) ⇒ ( )*

[24] ***CreateGroup*** *(access group name, access group type, full name) ⇒ (access group ID)*

[24] ***DeleteGroup*** *(access group name) ⇒ ( )*

[25] ***RenameGroup*** *(old name, new name) ⇒ ( )*

[25] ***AddToGroup*** *(access group$_1$, access group$_2$) ⇒ ( )*

[26] ***RemoveFromGroup*** *(access group$_1$, access group$_2$) ⇒ ( )*

[26] ***TranslateGroupName*** *(access group name) ⇒ (access group ID, full name)*

[27] ***TranslateGroupID*** *(access group ID) ⇒ (access group name, full name)*

[27] ***ListMembers*** *(access group name, transitive closure flag, list option flag) ⇒ (member list)*

[27] ***ListMembership*** *(access group name, transitive closure flag, list option flag) ⇒ (membership list)*

File Server Primitives

[28] ***OpenFile*** *(FID, queue request flag, mode, use transactions flag) ⇒ (IPC port)*

[29] ***ReadFile*** *(data unit count, position) ⇒ (data, data length, hole length)*

[30] ***WriteFile*** *(data, position) ⇒ ( )*

[31] ***PositionFile*** *(input offset, input offset base, output offset, output offset base) ⇒ (new input pointer, new output pointer)*

[31] ***TruncateFile*** *(new length) ⇒ ( )*

[32] *StartTrans (page list, queue request flag) ⇒ (transaction ID)*

[33] *AugmentTrans (page list, queue request flag, transaction ID) ⇒ ( )*

[33] *EndTrans (transaction ID, abort flag) ⇒ ( )*

[34] *CloseFile (abort flag) ⇒ (file modification date)*

[34] *CreateFile (storage characteristics, data characteristics, mutability, advisory file type, print name, archit size) ⇒ (IPC port, FID)*

[35] *GetHeader (FID, field mask) ⇒ (header fields)*

[36] *SetHeader (FID, field mask, fields) ⇒ (header modification date)*

[36] *GetAccess (FID, sys/aux flag) ⇒ (access control list)*

[37] *SetAccess (FID, sys/aux flag, rights list) ⇒ ( )*

[37] *CheckAccess (FID, access group list) ⇒ (rights mask)*

[38] *SetSem (FID, timeout period) ⇒ (current holder, expiration time)*

[38] *ResetSem (FID) ⇒ ( )*

[39] *PurgeFile (FID) ⇒ ( )*

Name Server Primitives

[40] *LookUpName (pathname) ⇒ (entry value)*

[40] *ScanNames (pathname, name status, previous name, name count) ⇒ (entry value list, actual name count)*

[41] *EnterName (pathname, {FID, IPC port, symbolic link, UserDefined type value, directory name}) ⇒ (new name)*

[42] *DeleteName (pathname) ⇒ ( )*

[43] *UndeleteName (pathname) ⇒ ( )*

[43] *ExpungeNames (pathname, entry names) ⇒ ( )*

[43] *ScanArchivedNames (pathname, start time, end time, name count) ⇒ (entry value list, actual name count)*

[44] *CreateDir (pathname) ⇒ ( )*

[45] *GetDirAccess (directory name, access control list) ⇒ ( )*

[45] ***SetDirAccess*** *(directory name, rights list) ⇒ ( )*

[46] ***CheckDirAccess*** *(directory name, access group list) ⇒ (rights mask)*

[46] ***GetDefAccess*** *(directory name, sys/aux flag) ⇒ (access control list)*

[46] ***SetDefAccess*** *(directory name, sys/aux flag, rights list) ⇒ ( )*

[47] ***CheckDefAccess*** *(directory name, access group list) ⇒ (rights mask)*

[48] ***ChangeDir*** *(directory name) ⇒ ( )*

References

[1] Almes, G. and Robertson, G.
An Extensible File System for Hydra.
In *Proceedings of the 3rd International Conference on Software Engineering.* IEEE, May, 1978.

[2] *DECSystem10 Technical Summary*
Digital Equipment Corp., Maynard, Mass., 1977.

[3] *VAX-11 Architecture Handbook*
Digital Equipment Corp., Maynard, Mass., 1979.

[4] *PDP-11 Processor Handbook*
Digital Equipment Corp., Maynard, Mass., 1979.

[5] Gray J.
Notes on Data Base Operating Systems.
Technical Report RJ2188(30001), IBM, February, 1978.

[6] Israel, J., Mitchell, J. and Sturgis, H.
Separating Data from Function in a Distributed File System.
In D. Lanciaux (editor), *Proceedings of the Second International Sysmposium on Operating Systems.*
IRIA, Rocqencourt, France, October, 1978.

[7] Lampson, B.W.
Atomic Transactions.
In *Lecture Notes for the Advanced Course on Distributed Systems -- Architecture and Implementation.*
Institut fur Informatik Technische Universtat Munchen, Munich, Germany, March, 1980.

[8] Metcalfe, R.M. and Boggs, D.R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM 9(7), July, 1976.

[9] Organick, E.I.
The Multics System: an Examination of its Structure.
MIT Press, Cambridge, Mass., 1972.

10] Rashid, R. F.
An Inter-Process Communication Facility for Unix.
Technical Report CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, March, 1980.

Ritchie, D. M. and Thompson, K.
The UNIX Time-Sharing System.
Bell System Technical Journal, July-August, 1978.

)epartment of Computer Science, Carnegie-Mellon University.
roposal for a Joint Effort in Personal Scientific Computing.
igust 1980.

icker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F. and Boggs, D.R.
: A personal computer.
nical Report CSL-79-11, Xerox Palo Alto Research Center, August, 1979.

[14] Thompson, M., Robertson, G., Satyanarayanan, M. and Accetta , M.
Spice File System.
Technical Report, Department of Computer Science, Carnegie-Mellon University, to appear 1980.

[15] White, R.M.
Disk Storage Technology.
Scientific American 243(2), August, 1980.